GitOps with Azure Kubernetes using ArgoCD

Problem Statement

Kubernetes is the Industry standard today to run application workloads with its orchestration capabilities like self healing, auto scaling etc.

Azure Kubernetes Service (AKS) is a **managed Kubernetes service** that allows users to deploy and manage containerized applications on Azure.

For every application running on Kubernetes platform, there are couple of components. One is **application source code** and other one is related to **cluster management.**

Continuous Integration part covers the **application** and **Continuous Delivery** part covers the **Cluster management**.

As part of **Continuous Integration**, when application source code commits in GitHub, automated CI pipeline pulls the code, builds, run test cases, generate Docker file and upload to Container registry. In addition, it updates the Kubernetes configuration file (deployment.yaml) with the latest Docker image version.

Continuous Delivery pipeline, deploys the updated Configuration files (deployment.yaml, service.yaml and others like configmap, secrets, namespace etc else helm charts) on Kubernetes Cluster. This is more of **push based approach** where the configuration files are applied on target Cluster with **kubectl apply** command.

Opportunity for improvement in this approach (non GitOps way of CICD) as follows:

- Even when there is no change in application code like updating replicas in deployment.yaml scenario as well, we have to execute both CI and CD jobs.
- ***Cluster credentials to be stored in a vault*** for CICD process to leverage and push changes. This is a ***security risk*** situation
- In Kubernetes deployments, we ***can't check for the successful completion of deployment***. Only we can execute the Kubectl apply command as part of deployment. We may need to execute test scripts to validate the completion of deployment

These areas of improvement items are handled in GitOps way of working enabled by tools like Argo CD, Flux. In this blog, we will be covering the Argo CD way of implementation of GitOps in Azure Kubernetes.

Azure Kubernetes come with "GitOps" feature leveraging Flux as its feature. Hence, I'm more Intrested in covering the "Argo CD" way of implementation. Audience of this blog may review "Flux" GitOps implementation provided by Azure Kubernetes which is much simpler.

A key information to note - **Argo CD is not a replacement for CD tools like Jenkins, Azure DevOps release pipelines** etc. Its an enabler for GitOps specifically for Kubernetes platform. Argo CD is a CD tool. CI process may continue to get executed in AS-IS way. Argo CD to apply the changes from manifest files. Hence, its a CD tool.

Solution/ Architecture

What is GitOps ?

GitOps is a way of implementing continuous deployment for cloud native applications, using git as a single source of truth for declarative infrastructure and applications. In a GitOps workflow, the desired state of an application and its infrastructure is stored in a git repository, and any changes to this desired state are made by committing changes to the git repository. An automated system, such as a Kubernetes operator or a continuous deployment tool, is then used to monitor the git repository and apply any changes to the actual state of the application and infrastructure.

GitOps benefits -

- Ability to track changes to the application and infrastructure, roll back changes easily, and collaborate with other team members using standard Git workflows.
- Allows for separation of concerns between development and operations teams, as the application and infrastructure configuration can be managed independently.

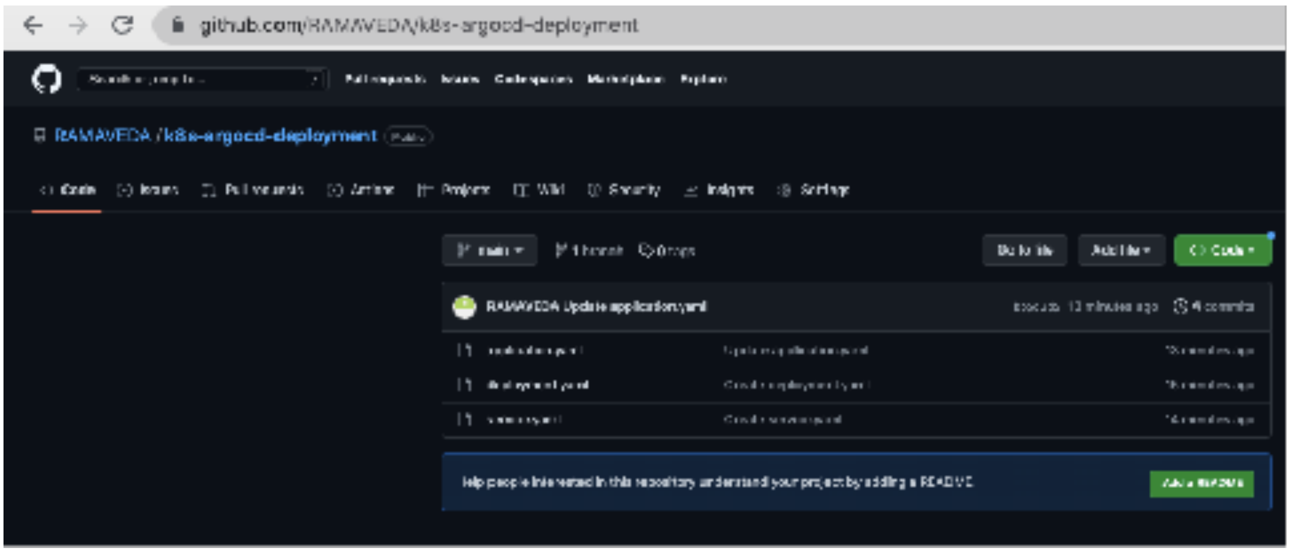
To implement GitOps in Azure Kubernetes, you will need to follow these steps:

1. Create a Kubernetes cluster on Azure using AKS.
2. Install and configure Argo CD on your AKS cluster.
3. Connect Argo CD to your Git repository, where you will store your declarative infrastructure and application configuration.
4. Define your application and infrastructure using Kubernetes manifest files and commit them to your Git repository.
5. Argo CD will automatically sync the desired state of your application and infrastructure with the actual state in your AKS cluster.

Argo CD is a pull based tool which picks the changes form GitHub rather pushing the code with kubectl commands

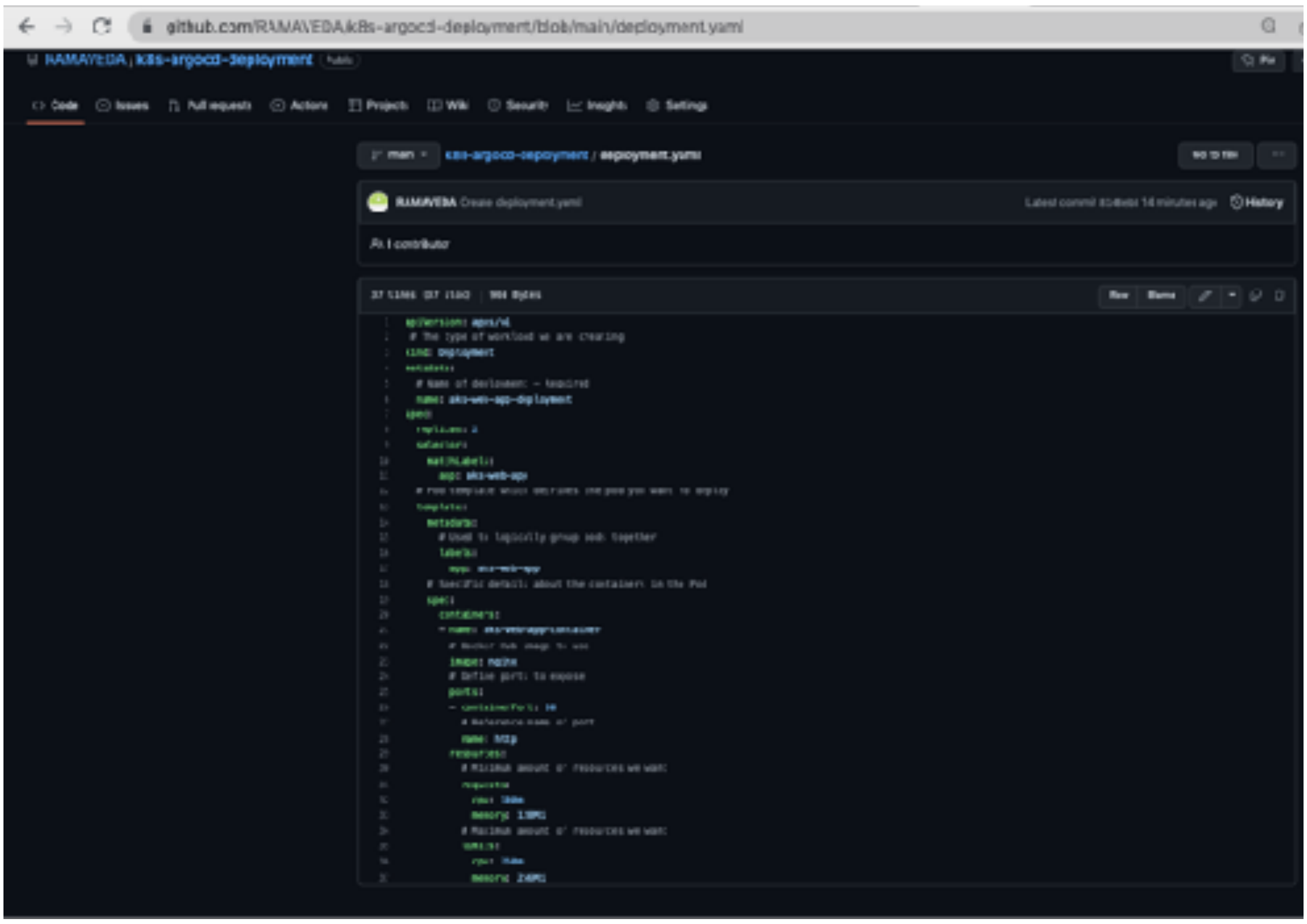
Technical Details and Implementation of solution

In order to implement GitOps with Argo CD in Azure Kuberentes, follow the below steps


github.com/RAMAVEDA/k8s-argocd-deployment
RAMAVEDA / k8s-argocd-deployment
Code
Issues
Pull requests
Actions
Projects
Wiki
Security
Insights
Settings
main
1 branch
0 tags
Go to file
Add file
Code
RAMAVEDA Update application.yaml
13 minutes ago
4 commits
Help people interested in this repository understand your project by adding a README.
Add a README

1. Setup GitHub

Create **deployment.yaml**, **service.yaml, application.yaml** files as noted below in GitHub repo

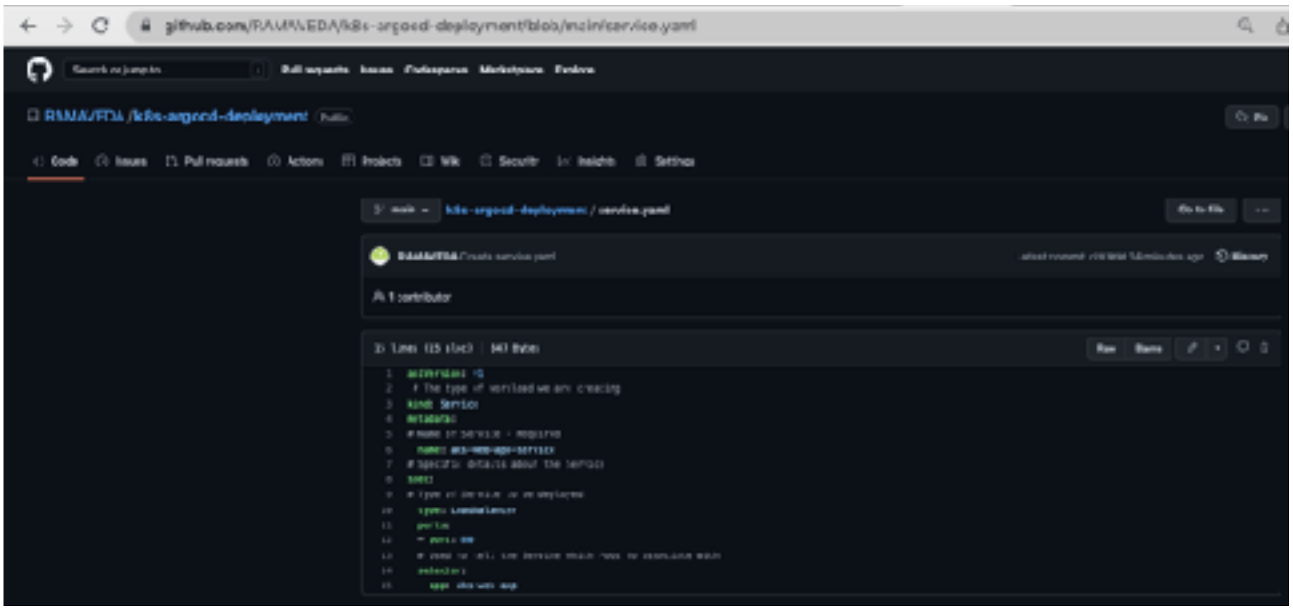

github.com/RAMAVEDA/k8s-argocd-deployment/blob/main/deployment.yaml
RAMAVEDA / k8s-argocd-deployment
Code
Issues
Pull requests
Actions
Projects
Wiki
Security
Insights
Settings
main
k8s-argocd-deployment / deployment.yaml
Go to file
RAMAVEDA Create deployment.yaml
14 minutes ago
History
1 contributor
37 lines (37 sloc) | 984 Bytes
Raw
Blame

```
apiVersion: apps/v1
# The type of workload we are creating
kind: Deployment
metadata:
  # Name of deployment - Required
  name: aks-web-app-deployment
spec:
  replicas: 2
  selector:
    matchLabels:
      app: aks-web-app
  # Pod template which decribes the pod you want to deploy
  template:
    metadata:
      # Used to logically group pods together
      labels:
        app: aks-web-app
    # Specific details about the containers in the Pod
    spec:
      containers:
      - name: aks-web-app-container
        # Docker Hub image to use
        image: nginx
        # Define ports to expose
        ports:
        - containerPort: 80
          # Reference name of port
          name: http
        resources:
          # Minimum amount of resources we want
          requests:
            cpu: 100m
            memory: 128Mi
          # Maximum amount of resources we want
          limits:
            cpu: 250m
            memory: 256Mi
```

deployment.yaml - Configuration file contain the container information like Docker image, replicas, resource limit etc. This defines the container information for Kuberenetes to orchestrate

Make a note in the below image, we have "2" replicas. Hence, when we deploy, we expect 2 PODS of NGINX to get deployed

service.yaml - Configuration file to expose the pod/container via Load Balancer service so that it can be accessed from Browser



github.com/RAMAVEDA/k8s-argocd-deployment/blob/main/service.yaml

RAMAVEDA / k8s-argocd-deployment

main / k8s-argocd-deployment / service.yaml

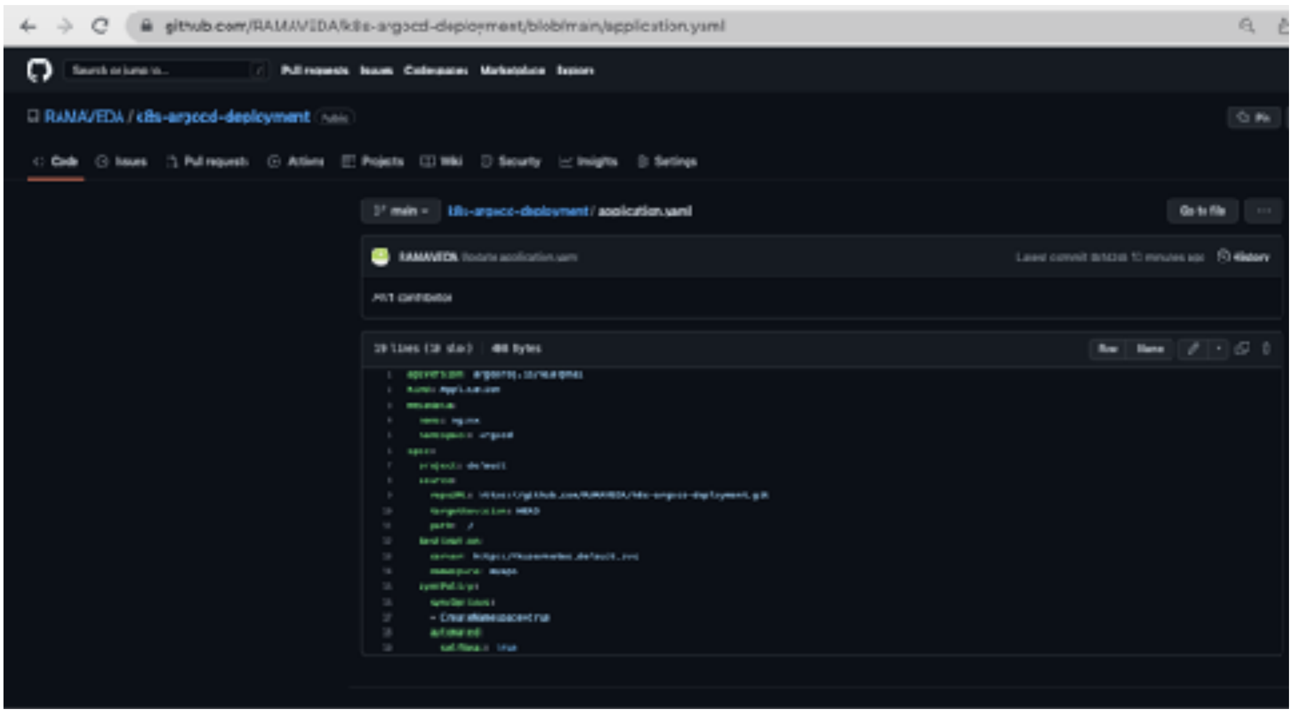
RAMAVEDA Create service.yaml

1 contributor

15 lines (15 sloc) | 340 Bytes

```yaml
apiVersion: v1
 # The type of workload we are creating
kind: Service
metadata:
# Name of service - required
  name: [illegible]
# Specific details about the service
spec:
# Type of service to be deployed
  type: LoadBalancer
  ports:
  - port: 80
  # [illegible]
  selector:
    app: [illegible]
```

application.yaml - Configuration file specifically for Argo CD implementation which is a bridge to integrate Argo CD with Git by configuring the path of GitHub repo, path of config files, etc.



github.com/RAMAVEDA/k8s-argocd-deployment/blob/main/application.yaml

RAMAVEDA / k8s-argocd-deployment

main / k8s-argocd-deployment / application.yaml

RAMAVEDA Update application.yaml

1 contributor

19 lines (19 sloc) | 488 Bytes

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: nginx
  namespace: argocd
spec:
  project: default
  source:
    repoURL: https://github.com/RAMAVEDA/k8s-argocd-deployment.git
    targetRevision: HEAD
    path: ./
  destination:
    server: https://kubernetes.default.svc
    namespace: myapp
  syncPolicy:
    syncOptions:
    - CreateNamespace=true
    automated:
      selfHeal: true
```

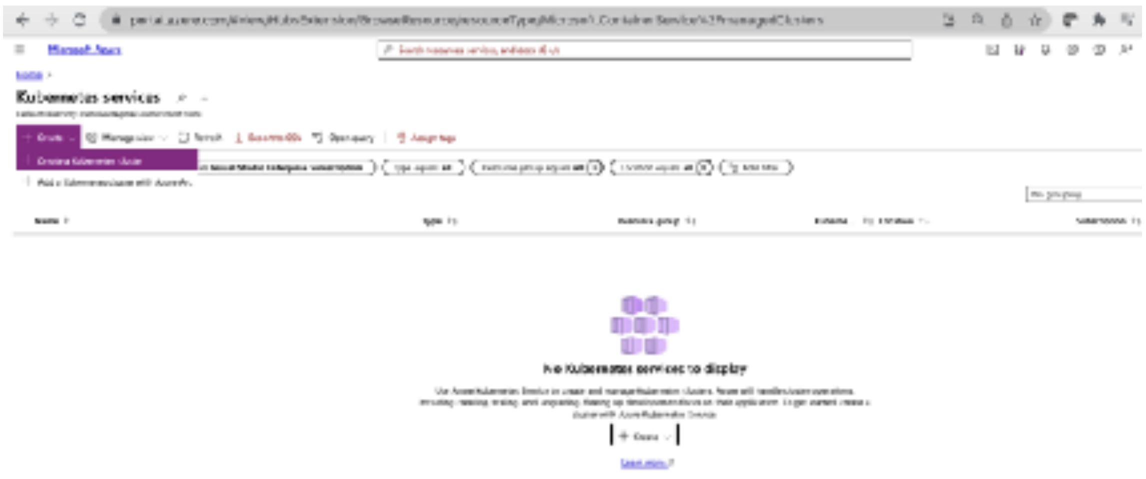
In this case, we have used "**nginx**" docker image for deployment.

In full blown production size applications, we may need to have couple of repositories. One for hosting the application source code and other repository for hosting the configuration files for Cluster management.In this demonstration, we focus on Cluster management (Continuous Delivery part) with Argo CD in AKS

2. Create Azure Kubernetes Cluster

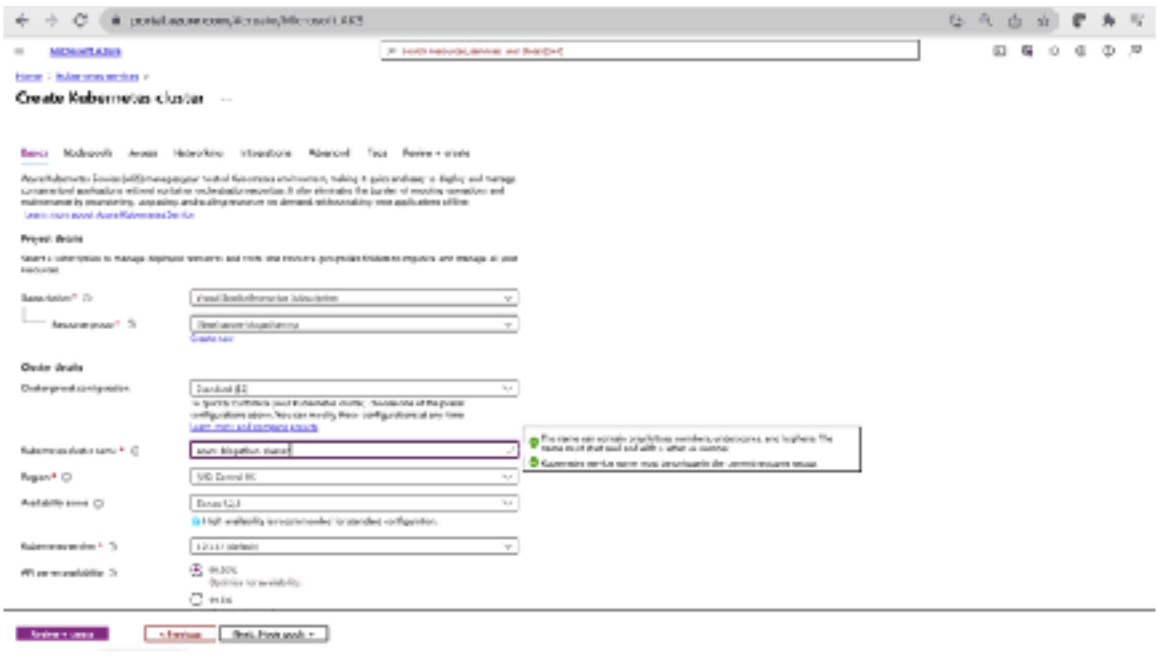
In order to create a Azure Kubernetes Cluster, we may need to have a subscription and resource group available as pre-requisite.

You may search for “Kubernetes” in the search of services and click create Kuberentes Cluster as noted below:



Upon click of “Create Kubernetes Cluster”, below form will open.
In the below image, you may absorb, we are creating the resource group as part of Cluster creation process. Also, you may need to provide the name for Cluster as well.

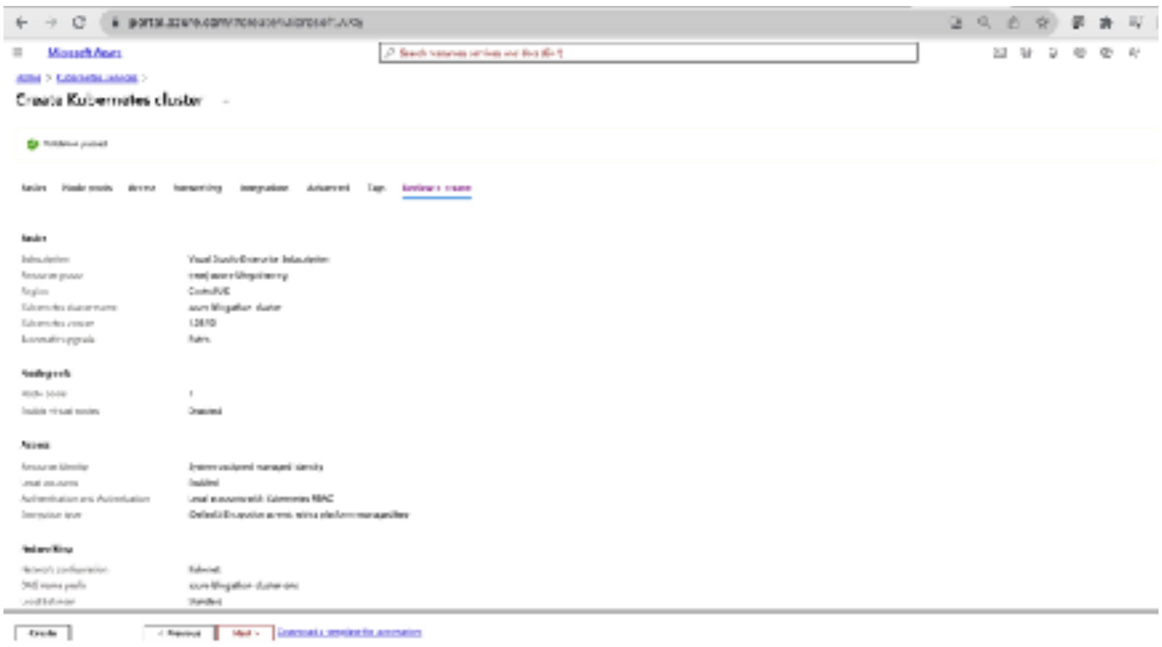
Post filling the form (across all tabs), you may click “Review and create”



Create Kubernetes cluster

Post

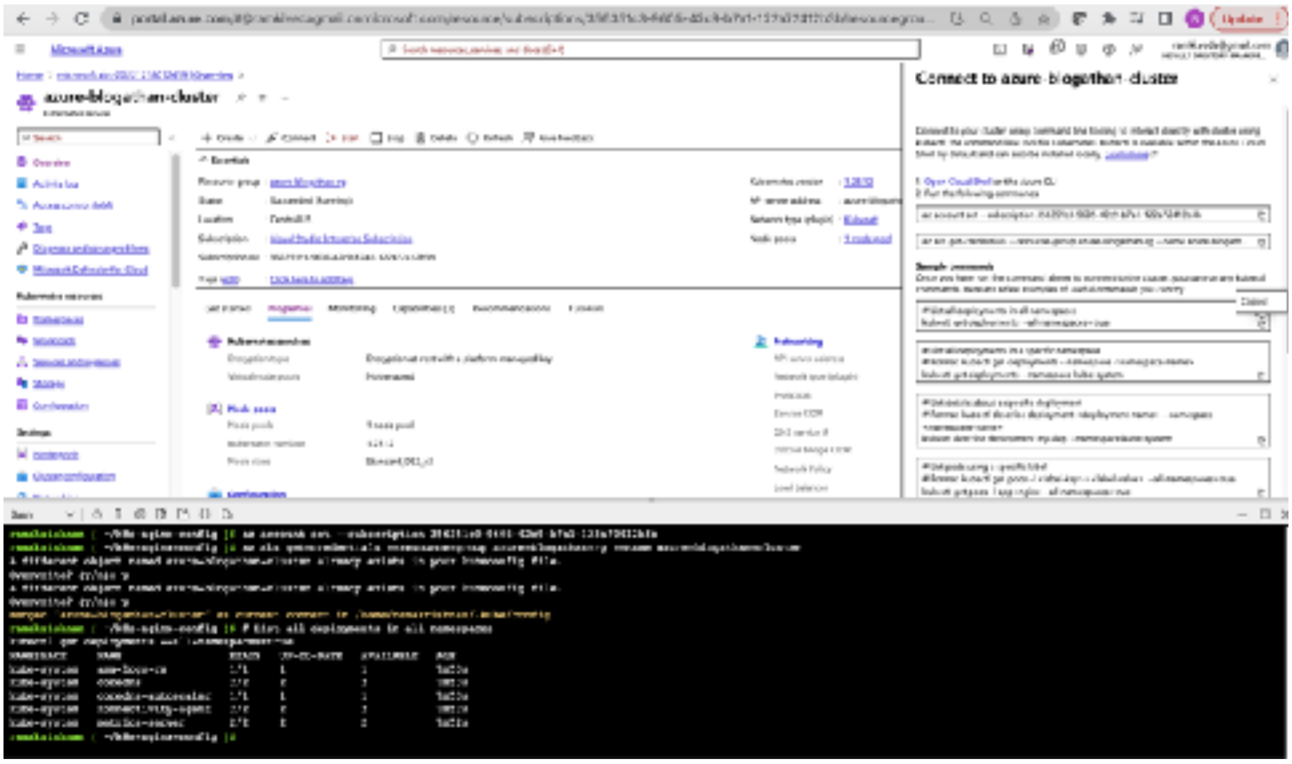
Validation pass, you may click “create” button to create the cluster as noted below:



Microsoft Azure

Create Kubernetes cluster

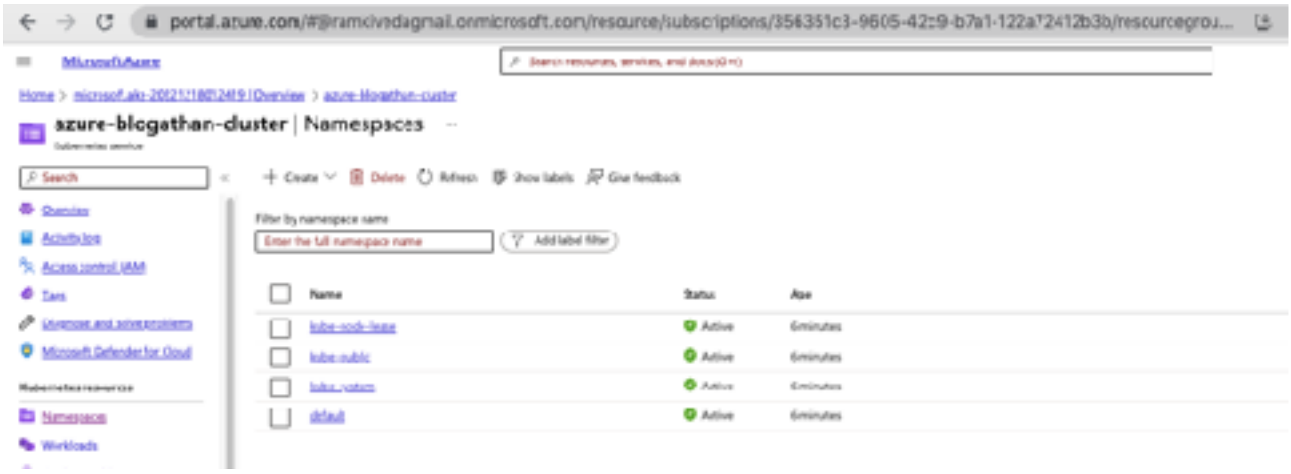
AKS Cluster as noted below will be created. You may next connect to the cluster.



Connect to azure-blogathan-cluster

```
[illegible]
```

As noted in below image, you may connect to the cluster by clicking, “connect” button. Commands will be shown on the right side. You may use the “Cloud Shell” to execute the commands as noted below to connect to cluster and check the “default” deployments.



azure-blogathan-cluster | Namespaces

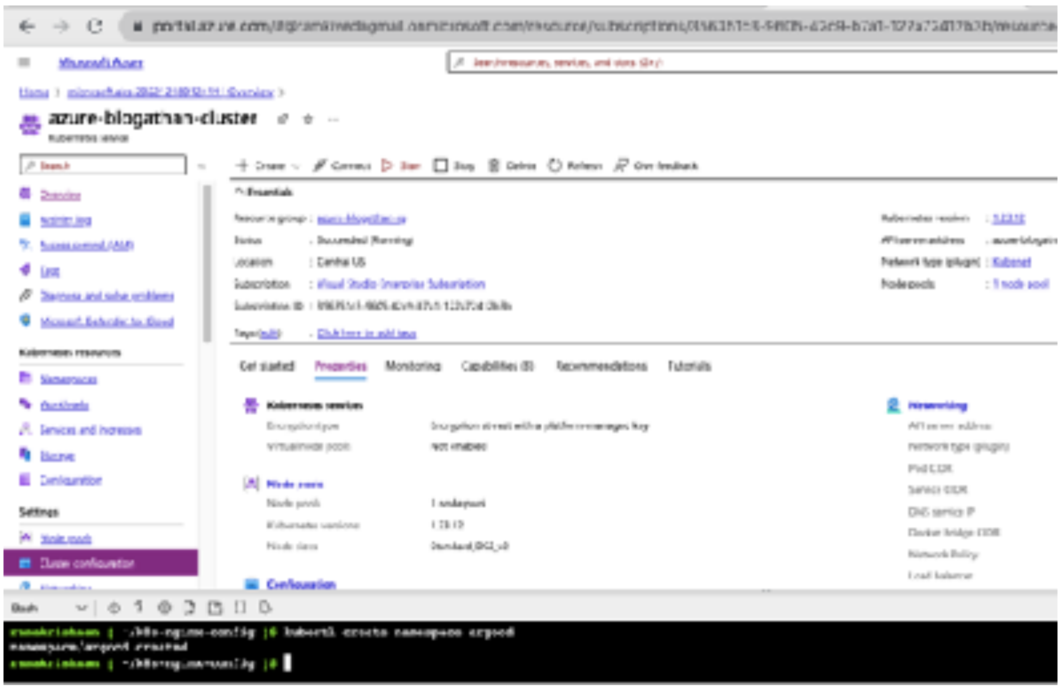
Name	Status	Age
kube-node-lease	Active	6minutes
kube-public	Active	6minutes
kube-system	Active	6minutes
default	Active	6minutes

In the below image, we can observe the default namespaces created by Azure Kubernetes when Cluster get created.

3. Install Argo CD in Azure Kuberenetes Cluster

1. Create a namespace in AKS cluster using the below noted command:

   kubectl create namespace argocd

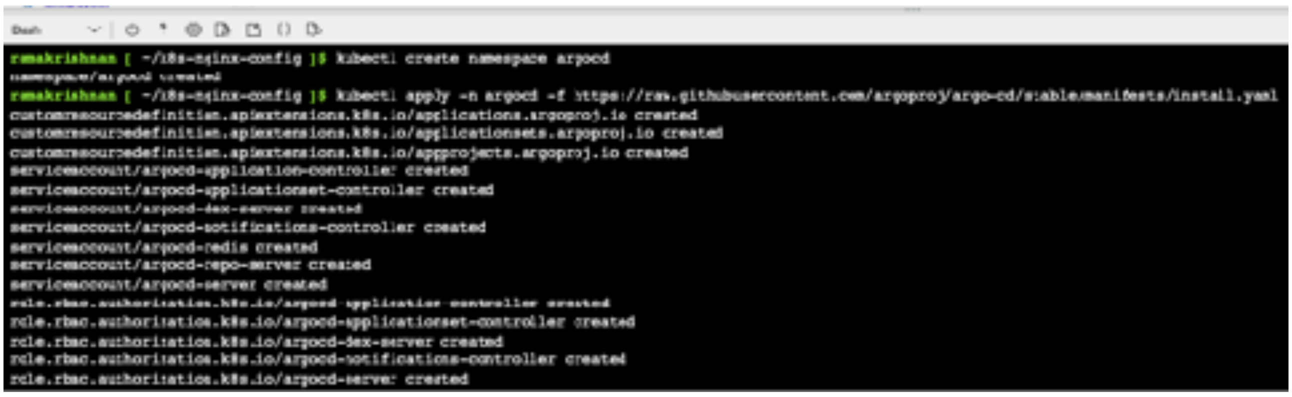


```
ramakrishnan [ ~/k8s-nginx-config ]$ kubectl create namespace argocd
namespace/argocd created
ramakrishnan [ ~/k8s-nginx-config ]$
```

This namespace will hold all Argo CD related objects

2. Install Argo CD in AKS cluster using the below noted command:

   kubectl apply -n argocd -f https://raw.githubusercontent.com/argoproj/argo-cd/stable/manifests/install.yaml



```
ramakrishnan [ ~/k8s-nginx-config ]$ kubectl create namespace argocd
namespace/argocd created
ramakrishnan [ ~/k8s-nginx-config ]$ kubectl apply -n argocd -f https://raw.githubusercontent.com/argoproj/argo-cd/stable/manifests/install.yaml
customresourcedefinition.apiextensions.k8s.io/applications.argoproj.io created
customresourcedefinition.apiextensions.k8s.io/applicationsets.argoproj.io created
customresourcedefinition.apiextensions.k8s.io/appprojects.argoproj.io created
serviceaccount/argocd-application-controller created
serviceaccount/argocd-applicationset-controller created
serviceaccount/argocd-dex-server created
serviceaccount/argocd-notifications-controller created
serviceaccount/argocd-redis created
serviceaccount/argocd-repo-server created
serviceaccount/argocd-server created
role.rbac.authorization.k8s.io/argocd-application-controller created
role.rbac.authorization.k8s.io/argocd-applicationset-controller created
role.rbac.authorization.k8s.io/argocd-dex-server created
role.rbac.authorization.k8s.io/argocd-notifications-controller created
role.rbac.authorization.k8s.io/argocd-server created
```

3. Change the argocd-server service type to LoadBalancer:

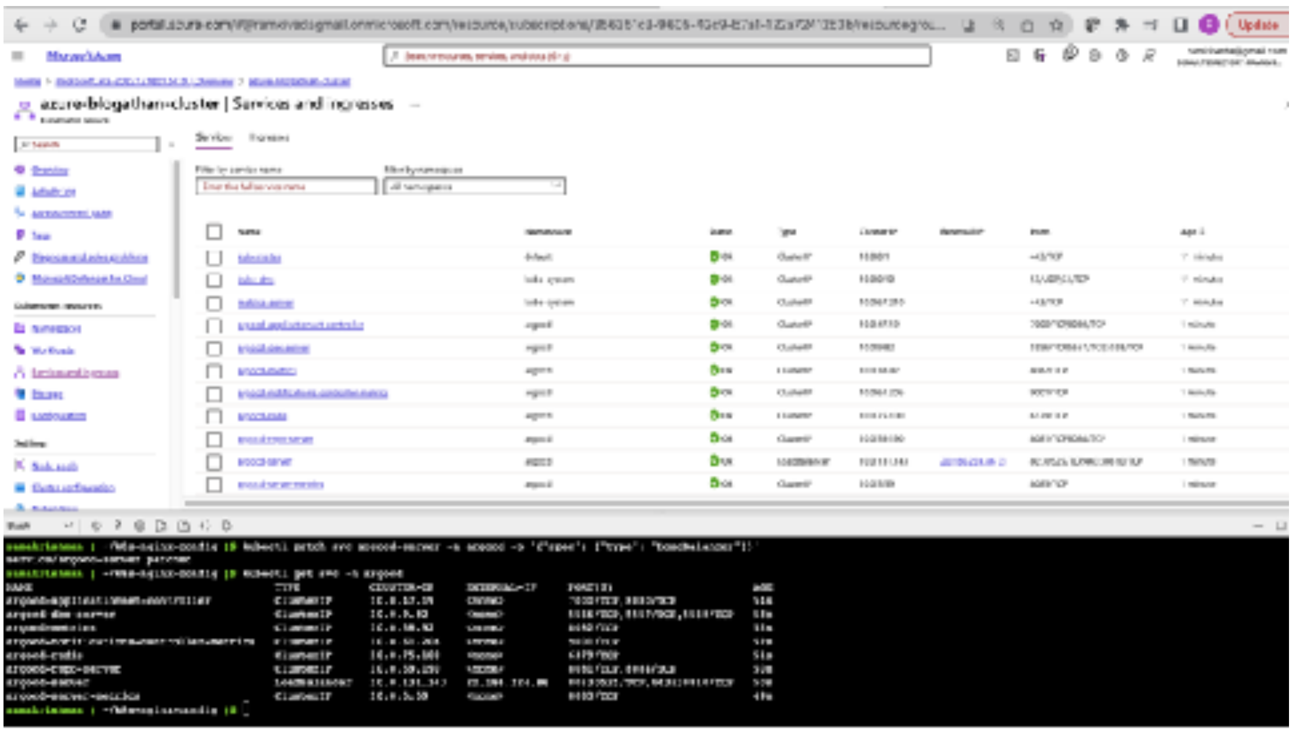
   kubectl patch svc argocd-server -n argocd -p '{"spec": {"type": "LoadBalancer"}}'

4. Access Argo CD:
   Now you will be able to see that the argocd-server service type has been changed to a LoadBalancer type. This means that it now has a public Azure load balancer attached to it with an external IP.

   kubectl get svc -n argocd

   Note: In production environments, use an ingress for the Argo CD API server that is secured. Above noted approach for exposing the Argo CD service is more for test/dev/demo exercises.

   Click the "external IP address" to launch Argo CD on browser as noted in below image:



5. Credentials to access Argo CD:

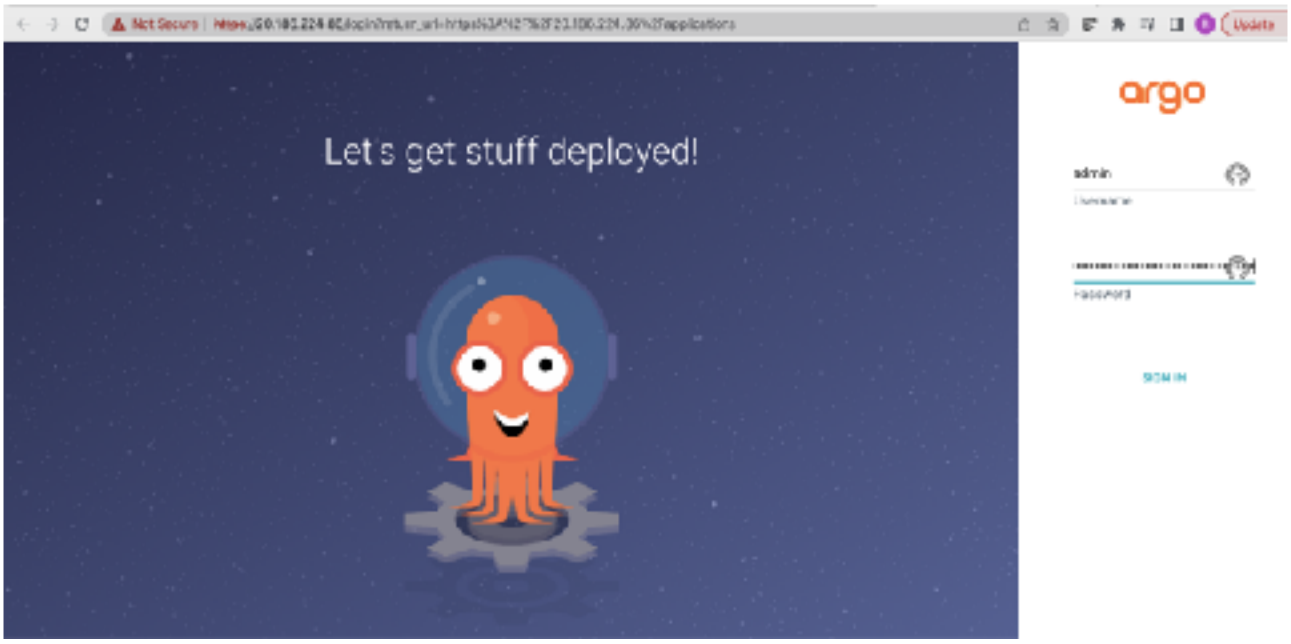
   Default User name: admin

   Password can be obtained by executing the below noted command:

   kubectl -n argocd get secret argocd-initial-admin-secret -o jsonpath="{.data.password}" | base64 -d; echo

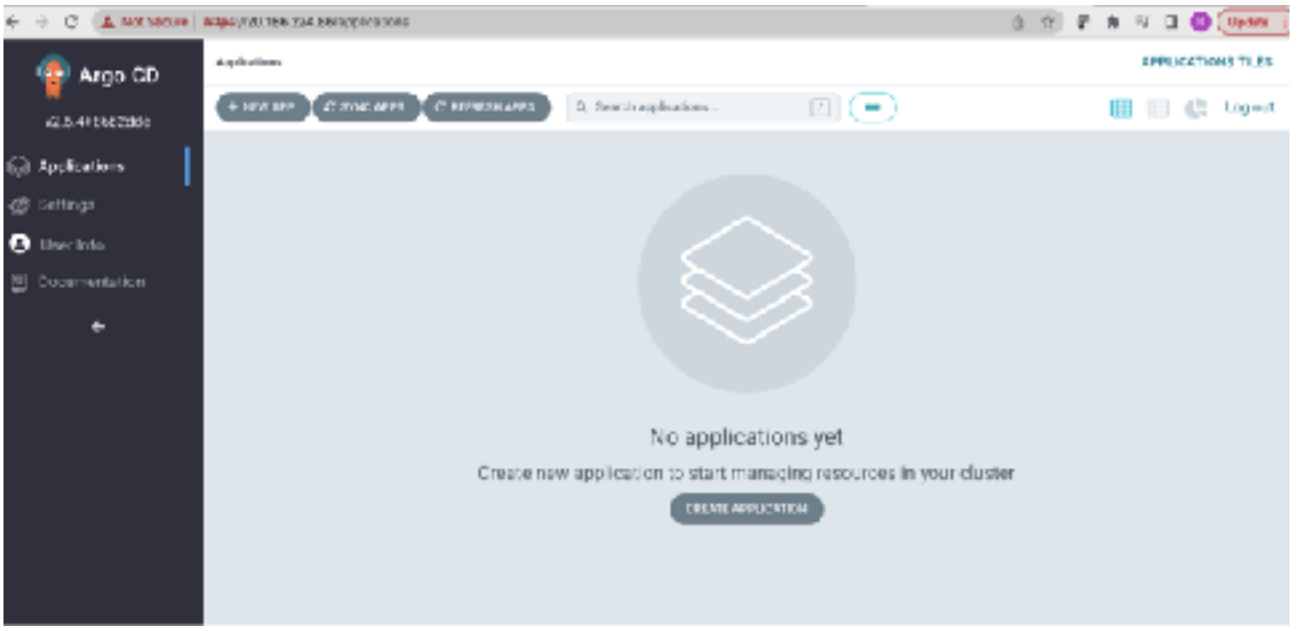
   Note: Warning message to launch Argo CD over browser may come. You may go-ahead and click the link to access the URL of Load balancer IP

6. Login to Argo CD as noted below:



Above noted image is the "Home" page of Argo CD. Since we are launching for the first time, no applications are listed.

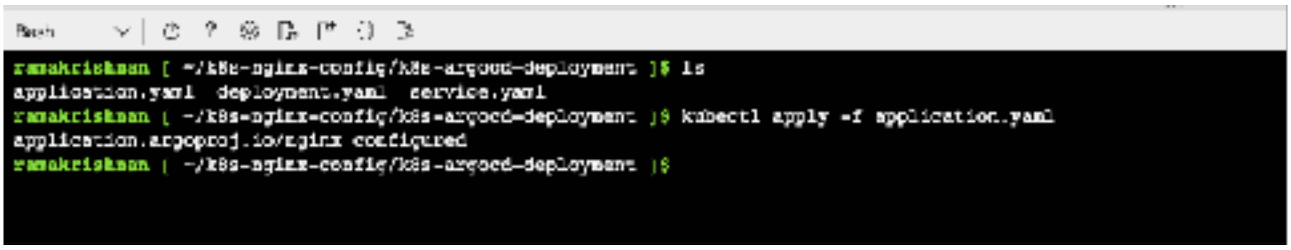
4. Establish connection between Argo CD and GitHub



In order to establish connection between Argo CD and GitHub, we may need "application.yaml" to be applied in Cluster

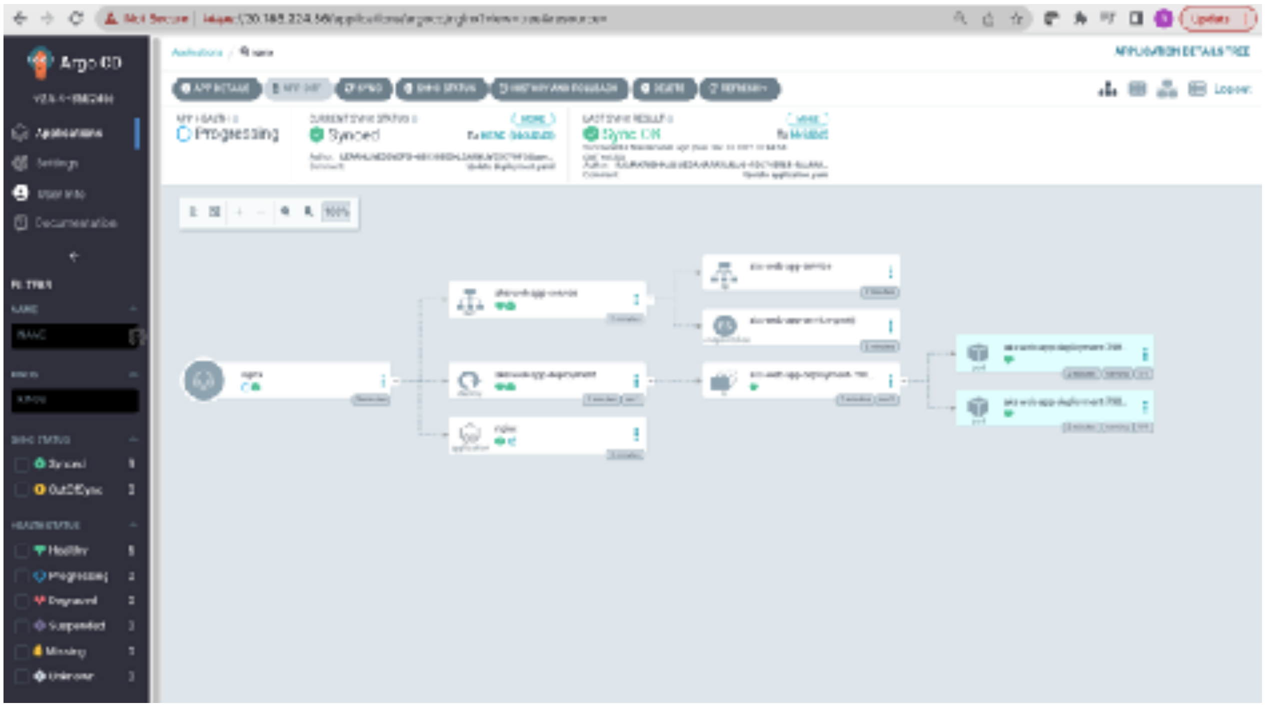
Clone the Git repository and navigate the application.yaml as noted below and apply application.yaml using:

Kubectl apply -f application.yaml



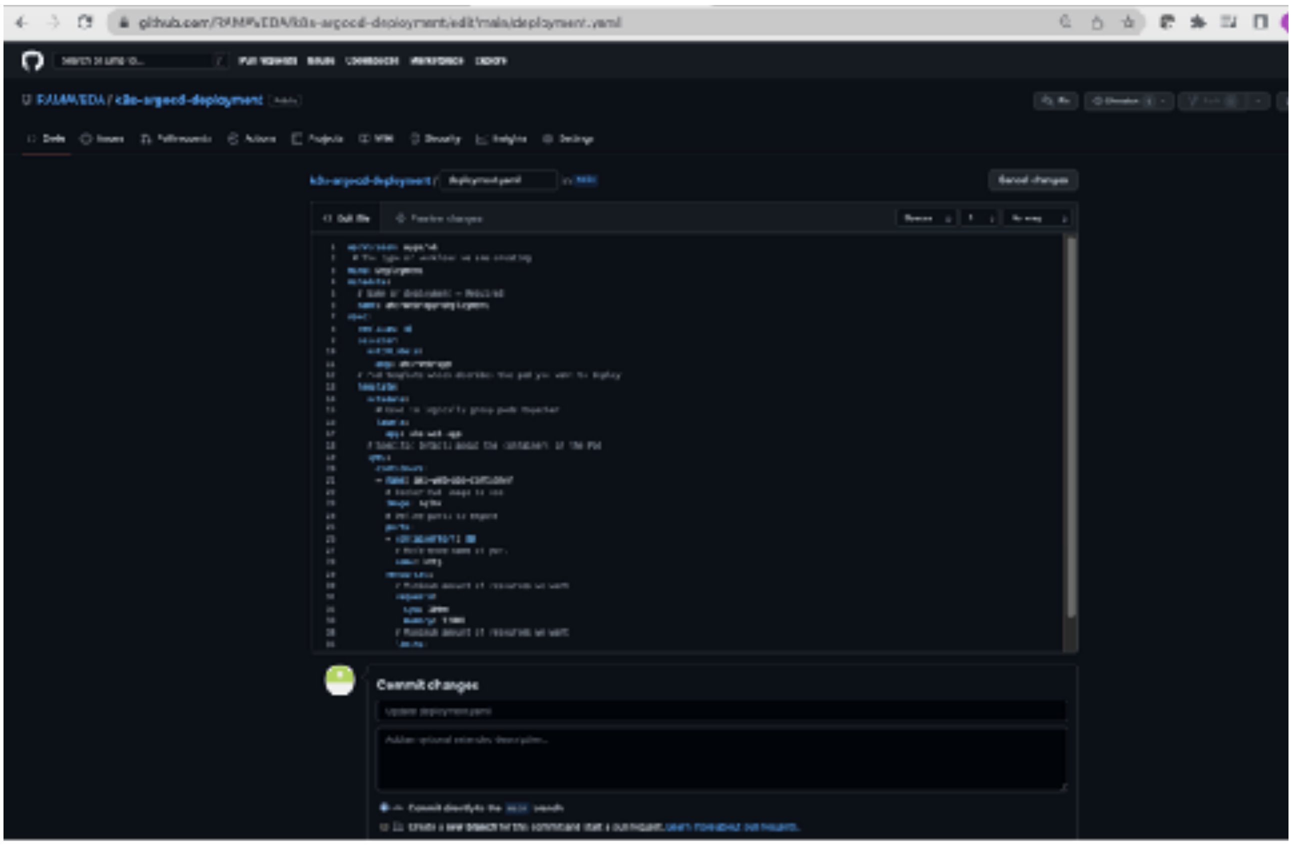
```
ramakrishnan [ ~/k8s-nginx-config/k8s-argocd-deployment ]$ ls
application.yaml  deployment.yaml  service.yaml
ramakrishnan [ ~/k8s-nginx-config/k8s-argocd-deployment ]$ kubectl apply -f application.yaml
application.argoproj.io/nginx configured
ramakrishnan [ ~/k8s-nginx-config/k8s-argocd-deployment ]$
```

As expected, as we have defined 2 replicas in deployment.yaml, we have 2 PODS running in Cluster post sync



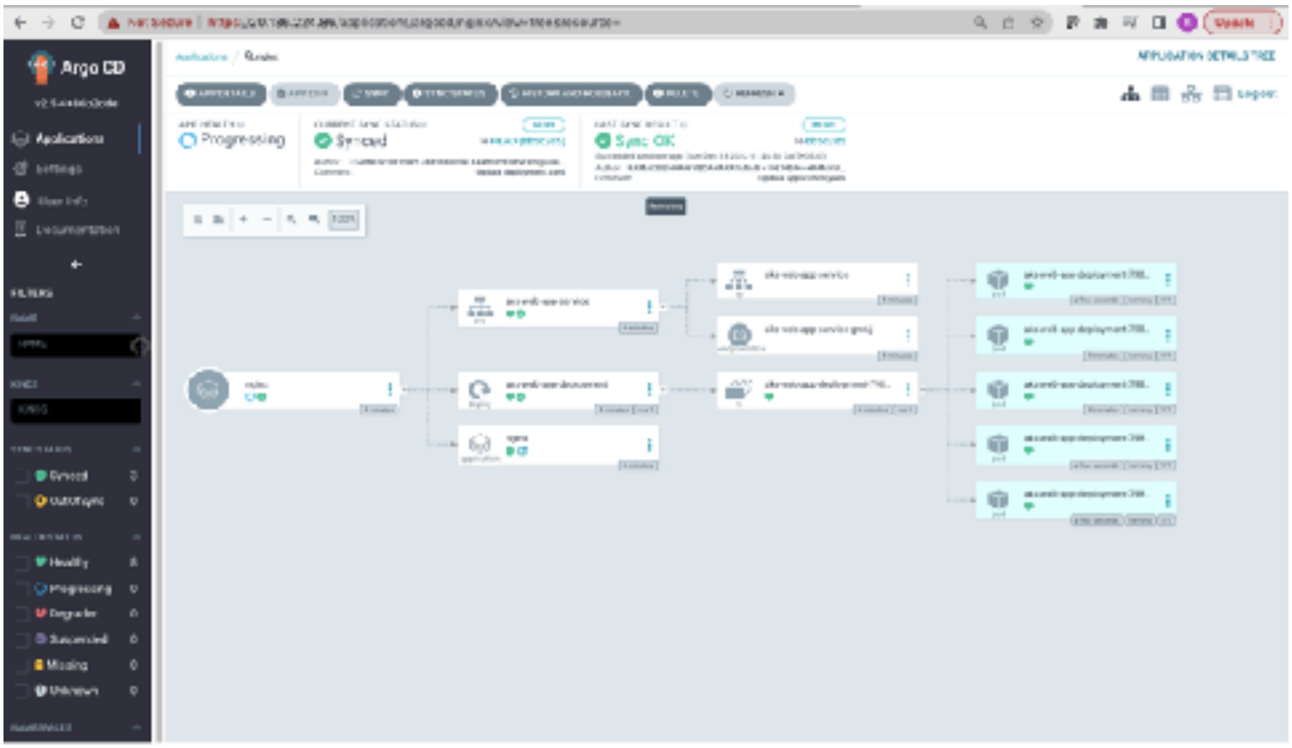
5. Argo CD synchronise the changes in GitHub configurations

Now, we are updating the replicas to "5" in deployment.yaml in GitHub and committing the change.


github.com/[illegible]/k8s-argocd-deployment/edit/main/deployment.yaml
k8s-argocd-deployment / deployment.yaml in main
Cancel changes
Edit file
Preview changes

```
apiVersion: apps/v1
 # The type of workload we are creating
kind: Deployment
metadata:
  # Name of deployment - Required
  name: [illegible]
spec:
  replicas: [illegible]
  selector:
    matchLabels:
      app: [illegible]
  # Pod template which describes the pod you want to deploy
  template:
    metadata:
      # Used to logically group pods together
      labels:
        app: [illegible]
    # Specific details about the container in the Pod
    spec:
      containers:
      - name: [illegible]
        # Docker Hub image to use
        image: nginx
        # Define ports to expose
        ports:
        - containerPort: 80
          # [illegible]
          name: http
        resources:
          # Minimum amount of resources we want
          requests:
            cpu: [illegible]
            memory: [illegible]
          # Maximum amount of resources we want
          limits:
```

Commit changes
Update deployment.yaml
Add an optional extended description...
Commit directly to the main branch
Create a new branch for this commit and start a pull request.



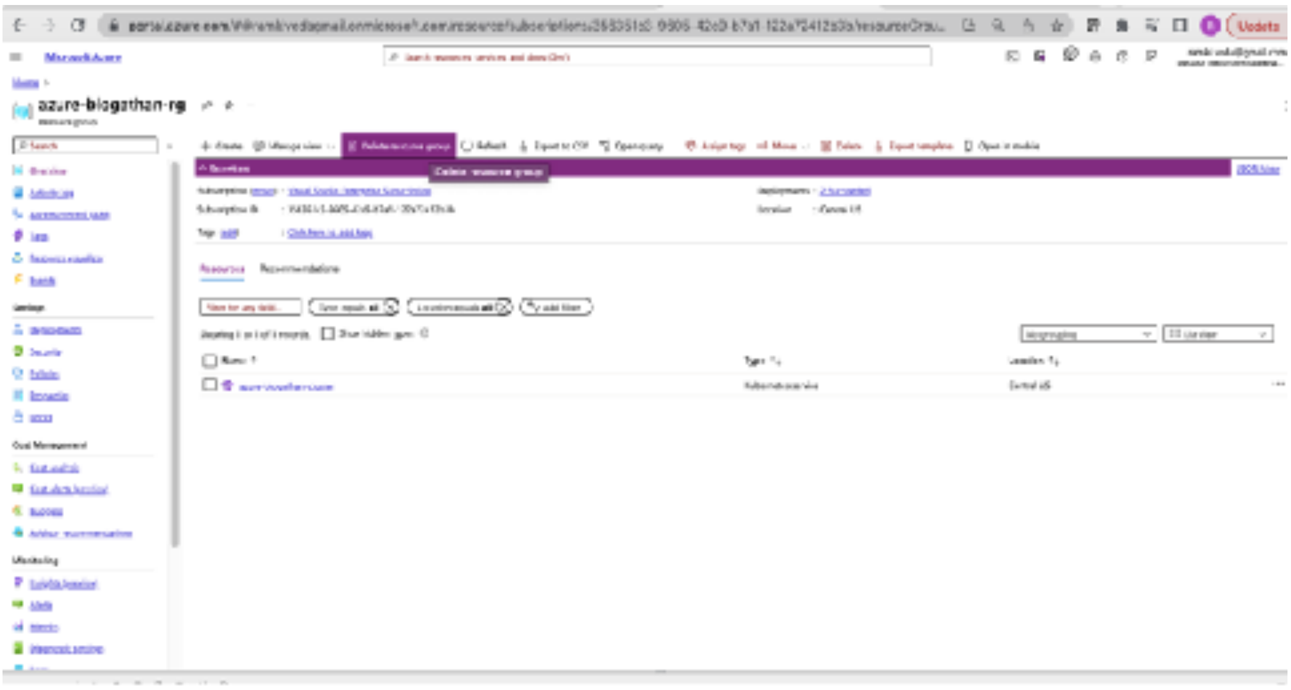
Page 12 of 15

Immediately in Argo CD, we can see the number of PODS refreshed as 5 as noted below

GitHub reference repository: https://github.com/RAMAVEDA/k8s-argocd-deployment.git

Clean up:

As Best Practise, you may need to clean up the setup to avoid unexpected cost incurred. As noted below, its always advisable to delete the "Resource Group" to ensure no unnoticed services run



Way forward:

Till now, we have experienced how Argo CD implements/syncronize the changes in Azure Kubernetes Cluster based on changes happen in GitHub repository. This is more focussed on Continuous Delivery.

As an extension of this demo, we can include the application code for Continuous Integration in a separate repository and build a CICD pipeline with GitHub actions

Create a new GitHub Actions workflow for your repository. The workflow should include the following steps:

Continuous Integration to include -

- Checkout the code from your repository.
- Build and test your application.
- Push the updated application image to a container registry, such as Docker Hub or Azure Container Registry.
- Update the Kubernetes manifest files in your repository with the new image tag, and commit the changes to the repository.

Continuous Delivery to include the below step -

- Argo CD will automatically detect the changes and deploy the updated application to your AKS cluster.

In addition, Azure Kuberentes Cluster and Argo CD installation can be automated with Terraform for Cluster creation source code in Git as well

Challenges in implementing the solution

Creating the synchronisation between Argo CD and GitHub as the application.yaml specification have to be accurate for establishing the sync

Business Benefit

Below noted Business Benefits are key outcomes of GitOps way of implementation.However its not just limited to below benefits.

Faster Time to Market: With GitOps way of deployment automation, we have integrated feedback control loop which speeds up mean time to deployment. This increases the Time to Market which is essential to gain Market share

Higher Developers Utilization: Limited experience is suffice for developers with Kuberentes as Git way of management eases out the Cluster management and application deployment as Git and Argo CD controls both application and Cluster management. Developers can focus on application development rather diluting the effort on Kubernetes

Improved stability, consistency: As Cluster management is automated, stability naturally improves. Also as both application and cluster configurations are managed in Git as single source of truth, consistency and standardisation of applications improve significantly.

Higher reliability: Ability to rollback and fork, enable teams gain stable and reproducible rollbacks which improved the mean time to recovery (reduced from hours to minutes)

Higher Security for Organizations:As the credentials of Cluster is no longer managed outside of Cluster, security of Cluster improves which enables true "DevSecOps" way of implementations with GitOps for Kuberenetes platforms